ARL-TR-7585 ● FEB 2016

US Army Research Laboratory

Integrating the Nqueens Algorithm into a Parameterized Benchmark Suite

by Jamie K Infantolino and Mikayla Malley

ARL-TR-7585 ● FEB 2016

US Army Research Laboratory

Integrating the Nqueens Algorithm into a Parameterized Benchmark Suite

by Jamie K Infantolino and Mikayla Malley
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE					*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE ***(DD-MM-YYYY)*** February 2016	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** May–August 2015
4. TITLE AND SUBTITLE Integrating the Nqueens Algorithm into a Parameterized Benchmark Suite		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Jamie K Infantolino and Mikayla Malley		**5d. PROJECT NUMBER** R.0006163.13
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIH-S Aberdeen Proving Ground, MD 21005-5067		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-7585
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
With heterogeneous computing increasingly a prevalent component in the science and engineering fields, benchmarking has become even more important for evaluation of each new architecture. The main issue is having a benchmarking code that uses sound benchmarking methodologies that accurately predict performance and provide a fair comparison between architectures. This technical report describes methodologies that have been developed at the US Army Research Laboratory (ARL). It examines the procedure on how to implement a new benchmark application program that will be added to the suite already in development. Also discussed is the algorithm, its relevance to US Army applications, its implementation, and the results from running it on the algorithm on the ARL Department of Defense Supercomputing Resource Center's High Performance Computers.

15. SUBJECT TERMS
optimization, benchmarking

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Jamie K Infantolino
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	28	**19b. TELEPHONE NUMBER (Include area code)** 410-278-7121

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

The authors wish to acknowledge the High-Performance Computing Modernization Program (GS04T09DBC0017) for contributing support of this work.

INTENTIONALLY LEFT BLANK.

1. Introduction

Supercomputing and High-Performance Computing (HPC) are vital assets in the science and engineering fields. Problems are larger and more complex; therefore, they require more computing resources with more compute capability. Problems that a decade ago were considered infeasible can now be solved in a matter of hours. Modern supercomputers are composed of multiple architectures—otherwise known as heterogeneous computers—working together to solve a problem as quickly as possible, provided the programmer has the skill to use all of the different components correctly and efficiently.

This has opened a field of research to examine how to best use these supercomputers. One major unknown is how the different computer architectures are going to perform with different applications. One approach is to use a program that will provide performance numbers for each available architecture to help predict performance and determine the best computer for each application. These are normally called benchmark programs, or just benchmarks.

Benchmark programs are defined as programs that are used to document performance on a computer—or a set of computers—to compare performance between them or to document performance of a particular application on a given architecture. Creating a mechanism to accurately predict performance is extremely difficult on a heterogeneous computer given the complexity of the different architectures present and the necessary interface that must be maintained between them.

Current and future trends in heterogeneous computers require that there must be a way to predict performance on these machines. There is a need for ways to compare machines to determine the best one to use for a given application. The goal of this work is to develop a benchmarking method to predict performance on heterogeneous computers through the use of optimal benchmark programs, while only maintaining one code base to minimize the amount of time required to maintain the benchmark programs. There are 2 benchmarking methods most commonly used.

One of the main problems when benchmarking HPC machines is comparing optimal performance across a variety of architectures. Usually this requires more than one code base to ensure optimal programs are run on each architecture, ensuring accurate and fair comparisons. Code optimized to run on one architecture is not guaranteed to run optimally on another architecture. Another issue benchmarking mechanisms face is how to choose programs that are reflective of real world scenarios to ensure the performance numbers produced can be correlated to real world performance. It is important to choose benchmark programs appropriately; they cannot be too big or too small, and they must be easy to understand, easy to maintain, easy to execute, and reflective of what will be faced within the applications.

The first method involves creating one program and running it on each machine to produce performance numbers comparable across all machines. The most well-known example of this is the Top 500 list, which uses one program—Linpack—to rank the top supercomputers in the world. There are some advantages to this approach. The first being it is easy to maintain because you are only having to maintain one program instead of a collection of programs. Additionally, having one program makes it easier to optimize for one architecture each time it is run. It is also easier to run because you only have to worry about executing one program; therefore, you do not have to be concerned about differences in setup. There are also some disadvantages to this approach. Since only one algorithm is used, it is only good for predicting performance for that algorithm. It really cannot be used to make generalized statements about the architecture. Very rarely are complex applications (i.e., those faced by the science and engineering fields) covered by one simple benchmark program; therefore, benchmarking methods like this cannot be used to make general statements about the architecture's performance in real-world applications. Another disadvantage is that having one algorithm makes it harder to compare across architectures and determine the impact parameters have on the architecture.

The second method is to use a suite of application programs to benchmark an architecture. The most common approach for using a suite is based on the work presented by the University of California, Berkeley.[1] Within their report, the concept of 13 computational dwarfs is presented where the 13 dwarfs represent the most commonly faced problems in the science and engineering fields. By using a collection of these dwarfs, one can adequately predict how the architecture will perform with real world science and engineering applications. It also provides a good way to compare performance across architectures. However, a downside to this method is the effort it takes to maintain a suite. Normally the suite contains numerous smaller programs for which an interface has to be designed and maintained.

One problem both of these methods face is how to ensure comparisons between architectures and machines are as accurate and fair as possible. Optimizations on one architecture are not necessarily optimizations on other architectures. In fact, sometimes they can have a negative impact on performance. This could potentially lead to misleading results in benchmarking. If one is comparing results that have been optimized for one architecture yet run on an architecture where those optimizations hurt performance, it will not give accurate comparison data.

A potential solution to this problem is to optimize each benchmark program to each architecture. This will provide the fairest comparison data possible. However, doing this manually is nearly impossible especially as the number of potential architectures and benchmark programs increase. An automatic method to sweep over all tunable parameters is preferred. This concept is called "autotuning", where the kernel parameters for each benchmark program are tuned without human intervention for each available architecture. For this work, an exhaustive search method will be used for the autotuning method to ensure that all parameter combinations are explored.

A benchmark suite was previously started at the US Army Research Laboratory (ARL) that implemented a handful of the dwarfs and used autotuning methodologies to compare optimal performance of various architectures. The work presented here builds upon this by adding the Backtrack Branch and Bound (BBB) algorithm to that suite.

The next 2 sections provide more information about existing benchmark suites and the available open-source suites. Sections 4–6 provide a detailed explanation of the BBB benchmark program that was added in this effort, along with a description of the method used to add this kernel to the suite. Results from running the particular kernel on the ARL Department of Defense Supercomputing Resource Center's (DSRC's) Supercomputers are presented in Section 7. Finally, conclusions and future work are discussed in Section 8.

2. Benchmark Suites

Use of a benchmark suite to accurately predict performance has been used within the research community for years. Researchers have taken this idea and created their own open-source benchmark suites. It is in the best interest of this research group to use these suites as a foundation. A complete discussion of the benchmark suites can be found in a previously published ARL report.[2] The takeaway from the ARL technical report examining each benchmark suite is that there are 2 mature benchmark suites to fit the needs of ARL: OpenDwarfs and the Scalable Heterogeneous Computing Benchmark Suite (SHOC). Each of these has advantages and disadvantages, as was previously documented.[2] These 2 suites will be summarized here for convenience.

2.1 OpenDwarfs

The OpenDwarfs suite was created by Virginia Tech.[3] The suite consists of 13 high-level applications spanning a variety of domains covered by the computational dwarfs presented in Berkeley's report.[1] Each dwarf implemented in the suite represents a communication pattern or particular computation that is common to important applications to the industry, which is the goal of using a benchmark suite to predict performance. The particular dwarfs have been programmed and are therefore not optimized for a particular architecture as much as possible. This particular suite implements the computational dwarfs explicitly from the Berkeley report.[1]

2.2 SHOC

SHOC was created by researchers at Oak Ridge National Laboratory and the University of Tennessee[4] with the purpose of testing the performance and stability of computer architectures through the use of performance and stress testing. Performance testing is used to measure system performance, while stress testing is used to determine any hardware issues that may impact performance. The benchmark suite itself is broken into 3 levels of tests. Level 0 stress tests the low-level hardware characteristics such as bandwidth tests. Level 1 is the implementation of the computational dwarfs. Level 2 is the real world applications. This suite has a few optimization techniques it uses to produce the most accurate timing results possible. First, it runs one kernel for a long time to increase the odds of seeing transient effects from the architecture—which is important when trying to predict real world runtimes. It also minimizes the data transfers so that pure runtime is accurately measured. This suite also contains additional tests that are not present in Berkeley's report.[1]

3. Related Works

OpenDwarfs and SHOC are only 2 of many benchmarking suites available. There is also Rodinia,[5] Graph500,[6] and Linpack.[7] Each of these provides a slightly different way to benchmark various architectures. The field of benchmarking is constantly evolving and changing to meet the demands and specifications of new architectures and applications. A good example of this is the High-Performance Conjugate Gradient (HPCG)[8] benchmark that was released in late 2013 to better represent the modern day applications computers face. It is presented with the Linpack benchmark in the Top 500 list. Even with this evolving nature of preexisting benchmark suites, a portable solution that is designed to assess current and emerging architectures is lacking.

The goal of this work is to create a benchmark suite that is optimized for each available architecture by using autotuning methodologies. Other researchers have used OpenCL[9] or CUDA to autotune code.[10] The work presented in both papers mainly focuses on Graphic Processing Units (GPUs) and autotuning kernels for one architecture. Our work is broader given we are concerned with every available architecture and true heterogeneous computing. The work from the papers supports the case that autotuning is an important factor in improving performance; each author saw an improvement in performance when compared to the untuned version of the code. Other authors have compared results across multiple architectures[11] without the use of autotuning.

Autotuning itself has been a topic of research in the industry. There are many different methodologies that could be employed. In this work, we use an exhaustive search method to ensure we get the optimal answer each time. Others have used different search methods. Abu-Sufah and Karim[12] used training data to classify which type of matrix the given input is; they then set runtime parameters based on that classification. They also changed the way the data was processed based on the training data. In this work, we are using an empirical method to tune our kernel. The kernel presented here is quite different than the one presented in their paper;[12] therefore, we could not reuse their training data. Autotuning has also been used in performance portable solutions.[9,13,14] Phothilimthana et al.[13] present a case for autotuning but does not create a way to automatically achieve this in the way our solution does. The work presented by Pennycook et al.[14] shows how performance can suffer when bad values are chosen for various parameters, which supports the claim that autotuning is needed. However, they concentrate on a Message Passing Interface (MPI)/OpenCL approach, whereas we are benchmarking using only OpenCL.

4. Backtrack Branch and Bound

The BBB algorithm is a way to search for a solution to a problem among a variety of potential solutions. This algorithm is used to solve various search and global optimization problems for large spaces that are not easily managed. In a nutshell, the BBB algorithm's goal is to extend a partial solution into a complete solution. The partial solution includes consistent values for some of the variables. The simplest of problems use the BBB algorithm, but the computing time is lengthy. There are distinct differences between the backtracking procedure and the branch and bound procedure; the backtracking part is used to find all probable solutions that are obtainable for the problem, it traverses tree by using the Depth First Search method, realizes it made a mistake and redoes the last choice by backing up, and the state space tree is searched until a solution is found.

The branch and bound part solves optimization problems, it is able to traverse a tree in any way—the Depth First Search or Best First Search—if a better optimal solution is found, the presolution is let go, the state space tree is searched as a whole to find the optimal solution, and lastly, it includes a bounding function.

According to researchers at the University of California at Berkeley, a technique is required to figure out which parts of the search space are not needed because they do not contain anything that is worth searching.[1] Branch and bound algorithms use the divide-and-conquer principle, which means that the space that is being searched is divided into smaller subregions that are known as branching, and the bounds are found on the solutions that are in the subregion.[1] Two important aspects of the BBB algorithm are the expansion and evaluation procedures. During the expansion method, one of the internal nodes is taken from the search tree and is used to create its children.[15] During the evaluation process a lower bound is computed on the cost of the solution.

Lastly, the lower bound will lead in directing whether a node should be expanded or the order in which nodes are supposed to be expanded.[15] The branch and bound algorithm has 4 basic operations: branching, evaluation, bounding, and pruning. During the BBB process a problem is selected, the problem is branched and split into subproblems. Once the subproblems are created they are evaluated. If the solution to the subproblem is infeasible, then it is pruned and the subproblem is deleted. If the solution to the subproblem is not infeasible, then the bounds of that solution are updated and the subproblem is added to the queue. Figure 1 shows a general flow for the BBB algorithm.

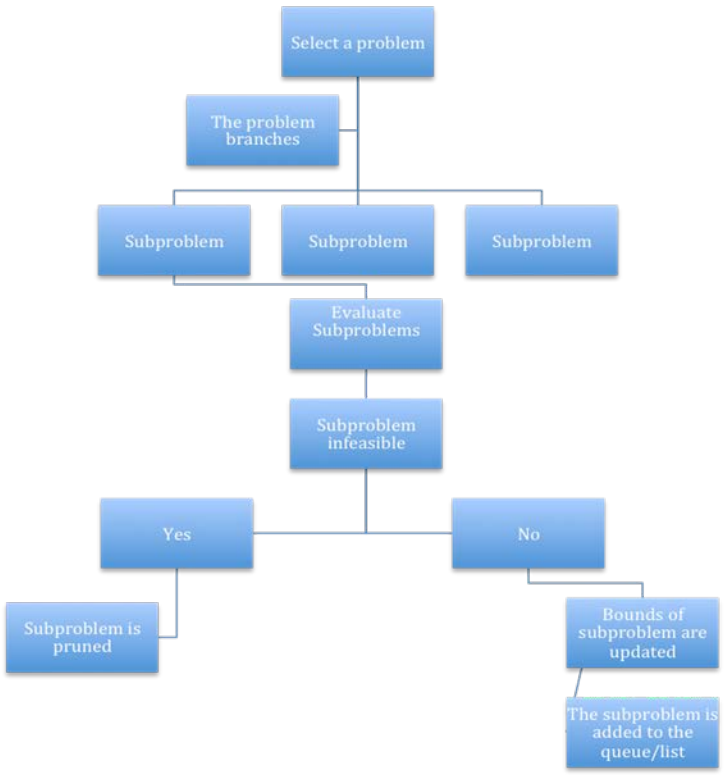


Fig. 1 Diagram of Nqueens algorithm

The good thing about backtracking is that it can repeat without difficulty through all variations of a set; in doing this, it guarantees accuracy by listing all options. For this to work, the search space must be pruned.[16] Thrashing is a disadvantage of the backtrack algorithm, which is a continuous failure for the same reason. If the problem cannot be identified, searches in different parts of the spaces will continue to fail.[16] Intelligent backtrack is used to fix this problem. The variable that continues to fail is returned immediately.

The 2 goals in the branch and bound search are to find an optimal solution and prove the problem's optimality. To reduce runtime, the only potential solutions that should be searched are those thought to be a good or possible optimal solution.[17] When only searching for the good, we will be able to prune the solutions that are of no use.

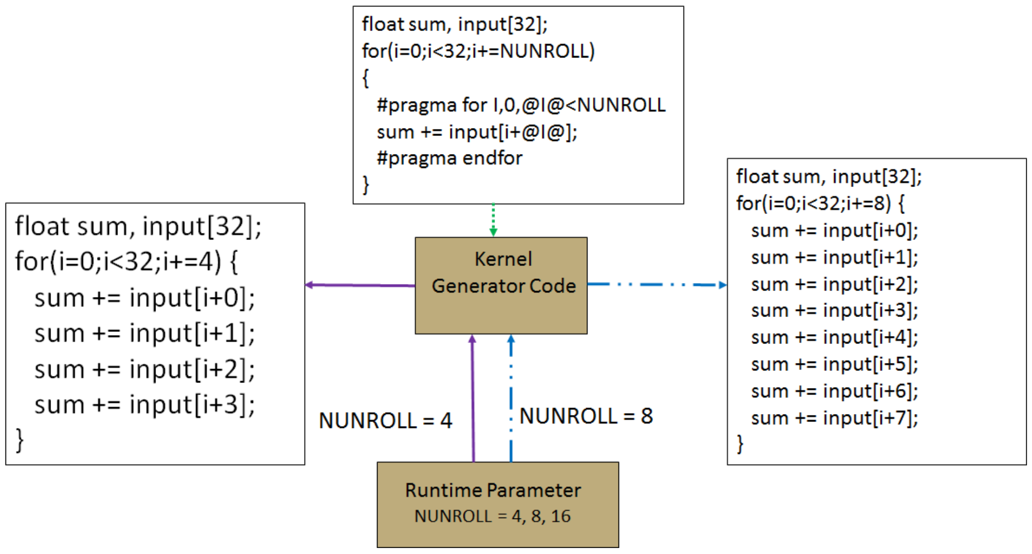
The BBB method can be parallelized. If the search space is split up so that each of the processors maintains their own subregion, the processors will be able to execute independently. If the processors cannot be kept occupied with high-quality work, it may be in the best interest to keep the tree small during the starting or ending point because the latter works more efficiently when the tree is smaller in size due to the parallel computation completing so much more work.[18] In a regular branch and bound search the active nodes may have to be thrown away because there is not enough space, which may contribute to the loss of an optimal solution. However, when using a parallel system it is expected to have enough memory so that the search can be completed.[18] To make sure that the search methods' parallel implementation is effective, it is a good idea to ensure that the search tree is consistently spread amongst all of the processors. If possible, all of the shared data should be placed close to each other; doing this will cut back on the time that the algorithm uses to travel and communicate.[15] The only issue is the fact that the search tree is only known during execution. Because of this, the algorithm has to be able to manage any tree that is created during execution time while keeping distribution and communication locality even.[15]

5. Autotuning Background

Producing meaningful benchmarking results across architectures is one of the challenges faced when creating a benchmark suite for heterogeneous computers. This is especially true when using a portable application program interface (API) such as OpenCL, which was used for this work. There are 2 types of APIs that could be used to write kernels: portable APIs (i.e., OpenCL) or vendor-specific APIs (i.e., CUDA). The vendor-specific APIs can be used to more precisely target the vendor-specific architecture associated with that API leading to greater performance. However, the same code cannot be used on different architectures from different vendors. OpenCL can be used across all architectures but the performance will tend to vary. OpenCL is portable but performance is not.

A significant aspect to benchmarking is code optimization, which ensures the results are as accurate as possible. However, optimizations can be misleading and lead to incorrect conclusions regarding the relative performance between architectures, especially on heterogeneous computers with immature technology. Heterogeneous computers have shown more sensitivity to source-level optimizations and runtime parameters than traditional computers. In this situation, the kernel can suffer from incorrect optimizations, resulting in unfair runtime comparisons between architectures, especially when a portable API is used. Fair runtime comparisons between architectures occur when the same level of optimization is present for each architecture. For a benchmarking methodology to use a portable API, optimization techniques must being integrated. The code can be manually tuned for each architecture but this should be avoided since it is prone to large differences in effort and uncertainty. Another way is to create a mechanism to automatically tune the code where different parameter values are swept to determine the best possible solution. In this work, source-level kernel parameterization along with an autotuning software developed in-house that sweeps over all possible combinations of kernel parameters was used to determine the optimal combination for each architecture so that the results for each architecture are the most optimal.

Figure 2 shows a high-level diagram of how the autotuning software works. This was created by researchers at ARL and used for this work. The core of the methodology is the kernel generator code, which generates variations of the code based on the combination of the input runtime parameters. A sample of this code can be seen in the box above the “Kernel Generator Code” box in the figure. This is a “C” preprocessing syntax code that takes in the input parameters seen in the lowest box as input and produces the code on the left and right depending on the input. In this particular case, the kernel generator code determines how many times to unroll the loop in the input code. The first input of “4” will be given to the kernel generator code and it will produce the results on the left, as demonstrated by the purple-line path. The same process occurs with an input of “8”, as seen with the dashed-blue lines, resulting in the code on the right.



```
float sum, input[32];
for(i=0;i<32;i+=NUNROLL)
{
  #pragma for I,0,@I@<NUNROLL
  sum += input[i+@I@];
  #pragma endfor
}
```

```
float sum, input[32];
for(i=0;i<32;i+=4) {
  sum += input[i+0];
  sum += input[i+1];
  sum += input[i+2];
  sum += input[i+3];
}
```

```
float sum, input[32];
for(i=0;i<32;i+=8) {
  sum += input[i+0];
  sum += input[i+1];
  sum += input[i+2];
  sum += input[i+3];
  sum += input[i+4];
  sum += input[i+5];
  sum += input[i+6];
  sum += input[i+7];
}
```

Fig. 2 Autotuning diagram

Within this work, multiple kernel parameters were used for the autotuning. Normally, the number of kernel parameters being swept is between 4 and 10. This produces a huge number of individual kernels that will be tested to determine the optimal kernel per architecture. As the proof-of-concept presented here, only 2 are swept. This is to minimize the data and ensure that the program and autotuning methods are working as intended.

6. Method

Previous work done by the research group at ARL integrated 6 kernels from the OpenDwarfs and SHOC benchmark suites into the in-house autotuning heterogeneous benchmark suite. These kernels include a reduction algorithm, a 2-dimensional (2-D) stencil algorithm, a Sparse Matric-Vector Multiplication (SPMV) algorithm, a Breath-First Search (BFS) algorithm, and a Fast Fourier Transform (FFT) algorithm. Each of these algorithms covers a different application area laid out by the University of California, Berkeley's technical report[1] on computational dwarfs. For each of these algorithms, a parameterized kernel was written and run through the autotuning code to obtain the optimal kernel for a variety of architectures. The dwarfs that were already covered are listed in Table 1 along with the general application covered.

Table 1 Dwarfs that have been implemented

Dwarf	Name	Description	Application area
Sparse linear algebra	SPMV	Matrix-vector multiplication with sparse matrices.	Embedded computing, general purpose computing, machine learning, graphics/games, Intel RMS
Spectral methods	FFT	Converts data from the time domain to the frequency domain to reduce the complexity of a calculation.	Embedded computing, machine learning, graphics/games
N-Body methods	N-Body	. . .	General purpose computing
Structured grids	Stencil2D	Calculate a 9-point 2-D stencil.	Embedded computing, general purpose computing, graphics/games
MapReduce	Reduction	Add a list of elements in an array via the use of reduction.	General purpose computing, machine learning, databases
Graph traversal	BFS	Traverse a graph using a breathe-first approach.	Embedded computing, machine learning, graphics/games, databases, Intel RMS

For this work, the remaining dwarfs were examined to determine the best application area to cover. To make this determination, various factors were considered, the first being the application area a potential dwarf covered and how it related to the others that were already covered. Ideally we wanted to pick one very different from the others that were already implemented to expand the breadth of coverage of the benchmark suite. The second consideration was whether the dwarf would be representative of a known US Army application. One of the end goals of this suite is to help the US Army predict the performance of US Army-relevant applications on the computers available.

The dwarfs that were not already covered are listed in Table 2 along with the general applications covered, as determined in Berkeley's report.[1]

Table 2 Dwarfs that have not been implemented

Dwarf	Description	Application area
Dense linear algebra	Data is stored in dense matrices or vectors.	Embedded computing, general purpose computing, machine learning, databases, Intel RMS
Unstructured grids	Consists of an irregular grid where data locations are selected and the data must be updated all together.	Machine learning, Intel RMS
Combinational logic	Logical functions are implemented and state is stored.	Embedded computing, machine learning, databases
Dynamic programming	Calculates a solution by breaking up the problem into smaller, overlapping problems.	Embedded computing, general purpose computing, machine learning, databases
Backtrack, Branch and Bound	Finds the optimal solution by breaking the problem space into smaller regions and removing suboptimal problems.	General purpose computing, machine learning
Graphical models	Nodes represent random variables and edges are conditional dependencies within a graph.	Embedded computing, general purpose computing, machine learning
Finite-state machines	Behavior is defined by states, transitions are defined by the input, and current state and events are represented by the system.	Embedded computing, general purpose computing, graphics/games

After examining these 2 tables, it was clear that all of the application areas were at least covered by a kernel already present in the benchmark suite. Therefore, the various applications areas themselves were examined to determine if there was an area that particularly matched up to a US Army-relevant application; this leads to machine learning as the application area chosen to explore. The Berkeley report[1] highlights 2 kernels that were added specifically to support machine learning: dynamic programming and BBB. The latter was chosen as the dwarf to concentrate on for this work. Either could have been chosen but the BBB provided a better opportunity to link performance to machine learning.

The first step was to examine what the other benchmark suites did for the BBB algorithm. OpenDwarfs implemented an Nqueens algorithm, which seemed to be a very well-known and respected BBB algorithm. It was chosen as our baseline algorithm as well. Then, a literature search was performed to determine if optimizations others made could be reused.

An Nqueens test bench program with autotuning methodologies was then developed. This included implementing an Nqueens kernel within OpenCL and creating a driver program to test the kernel. The results were documented as a baseline for the next stages of integration. Testing was also done at this time. It is important for the results to be accurate so that the timing results are valid.

The test bench program with 2 autotuning parameters was implemented, the first being the problem size, which was the size of the board, and second, the number of threads, which is the number of threads that were all run at once. The results were gathered on Excalibur and FOB to show how performance varied with the parameters.

7. Results

The Nqueens kernel was run on 2 heterogeneous machines at the ARL DSRC: FOB and Excalibur. The FOB is a 64-node heterogeneous cluster consisting of 16-IBM dx360M4 nodes, each with one NVIDIA Kepler K20M GPUs and 48-IBM dx360M4 nodes, and each with one Intel Phi 5110P. Each node contains a dual-Intel Xeon E5-2670 Central Processing Units (CPUs), 64-GB of memory and a Mellanox FDR-10 InfiniBand host-channel adapter. Excalibur has 3,13-Intel Xeon E5-2698 compute nodes with 128-GB of memory and an InfiniBand interconnect; 32 of these nodes have 256-GB of memory and an NVIDIA Tesla K40 GPU. More details on Excalibur can be found on the US Army DSRC website.[19]

Figures 3 and 4 show the results of FOB for the Many Integrated Core (MIC) processor and the GPU, while Tables 3 and 4 show the runtimes. It can be seen that overall the results on the GPU are slower than on the MIC; however, both have the same best performing kernel for the largest board. This kernel occurs when the number of threads executing at once is 16. It can also be seen that the lines on the graph begin showing more variation more quickly on the GPU than on the MIC. This can be interpreted as the GPU being more sensitive to the number of threads sooner than the MIC.

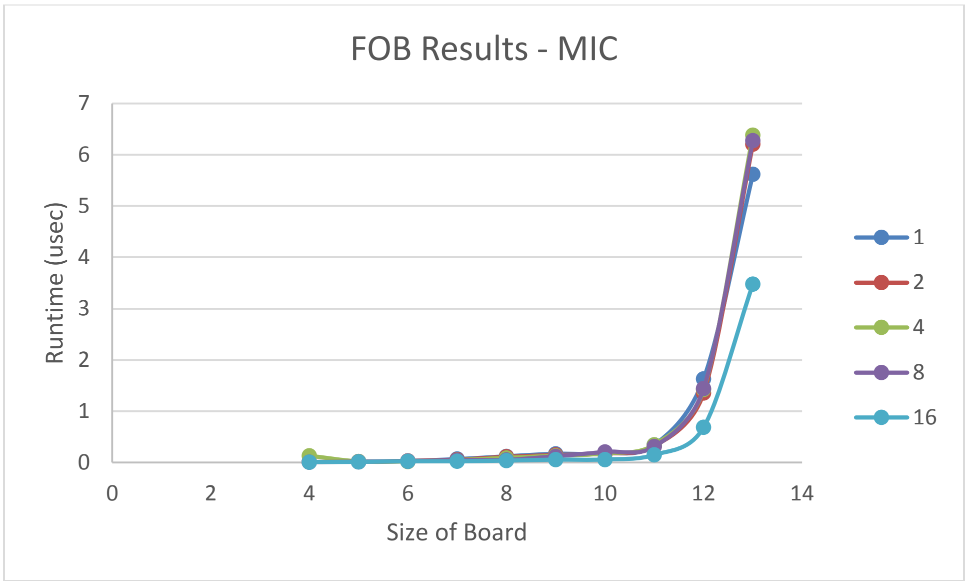


Fig. 3 Nqueens results on FOB using the MIC architecture

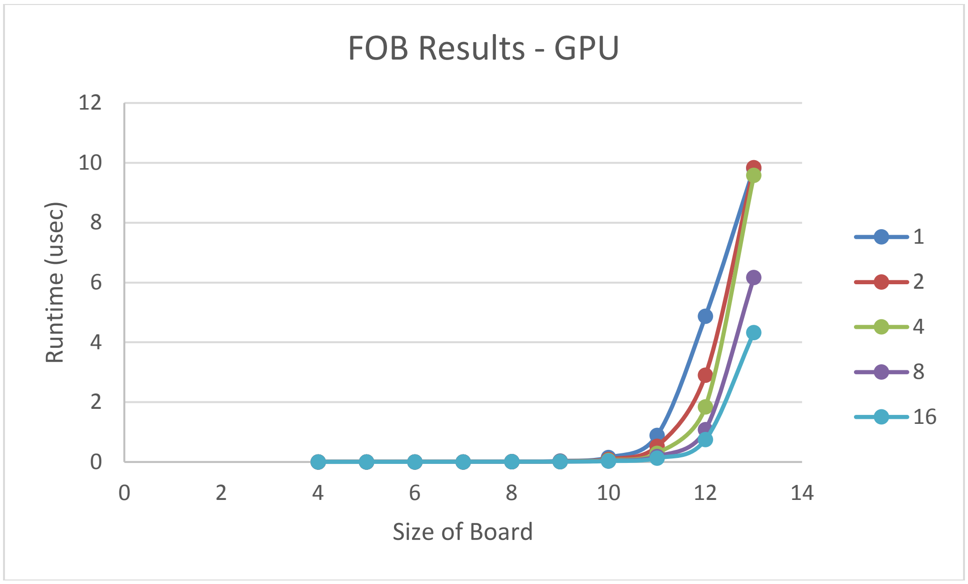


Fig. 4 Nqueens results on FOB using the GPU architecture

Table 3 Runtime in µs results from FOB using the MIC architecture

Size of board	N threads				
	1	2	4	8	16
4	0.00922333	0.010555	0.1284	0.00778667	0.00715833
5	0.020015	0.017565	0.01851	0.0150917	0.00976667
6	0.031845	0.03124	0.0186283	0.0221533	0.022465
7	0.06589	0.05767	0.04447	0.0441065	0.0218433
8	0.121363	0.111075	0.088975	0.0516883	0.0352317
9	0.169782	0.146932	0.129743	0.120732	0.05366
10	0.175228	0.176315	0.175922	0.206492	0.0574817
11	0.343343	0.32064	0.346177	0.310695	0.147945
12	1.63164	1.35784	1.42274	1.44006	0.686292
13	5.62035	6.21	6.38215	6.27627	3.48108

Table 4 Runtime in µs results on FOB using the GPU architecture

Size of board	N threads				
	1	2	4	8	16
4	0.00096167	0.00107833	0.00099167	0.00094833	0.00124
5	0.00142333	0.00144667	0.00161167	0.00170667	0.001785
6	0.002135	0.00235	0.00256	0.00255833	0.00266667
7	0.00297167	0.00351333	0.00395167	0.00399667	0.00429
8	0.00719833	0.00606333	0.00537	0.00588	0.0062
9	0.02837	0.01898	0.0137283	0.0109267	0.00946
10	0.146675	0.08932	0.0551933	0.0361517	0.027215
11	0.885253	0.52041	0.287963	0.185977	0.123232
12	4.86937	2.90203	1.84538	1.07812	0.746093
13	9.81857	9.83973	9.57822	6.161961	4.32638

The results from Excalibur are seen in Table 5 and Fig. 5. The figure has the same pattern as the figures from FOB. However, there are slight variations. When compared to the GPU results from FOB, the GPU on Excalibur has the same best kernel (i.e., N threads = 16), but there is more separation of the lines at the higher workloads. In addition, the worst performing workload performs far worse on this GPU than on the GPU from FOB.

Table 5 Runtime in µs results on Excalibur using the GPU architecture

Size of board	N threads				
	1	**2**	**4**	**8**	**16**
4	0.00171	0.00171	0.00175	0.00241	0.00172
5	0.00225	0.00248	0.00248	0.00257	0.00272
6	0.00301	0.00326	0.00333	0.00371	0.00469
7	0.00425	0.00458	0.00476	0.00514	0.00544
8	0.00906	0.00729	0.00674	0.00726	0.00771
9	0.03004	0.02128	0.01599	0.0128	0.01142
10	0.15133	0.09297	0.0593	0.03973	0.03022
11	0.91115	0.53566	0.31445	0.19191	0.12672
12	6.02307	3.48799	1.4461	1.16836	0.75468
13	10.3828	17.6849	6.92684	7.0915	4.58497

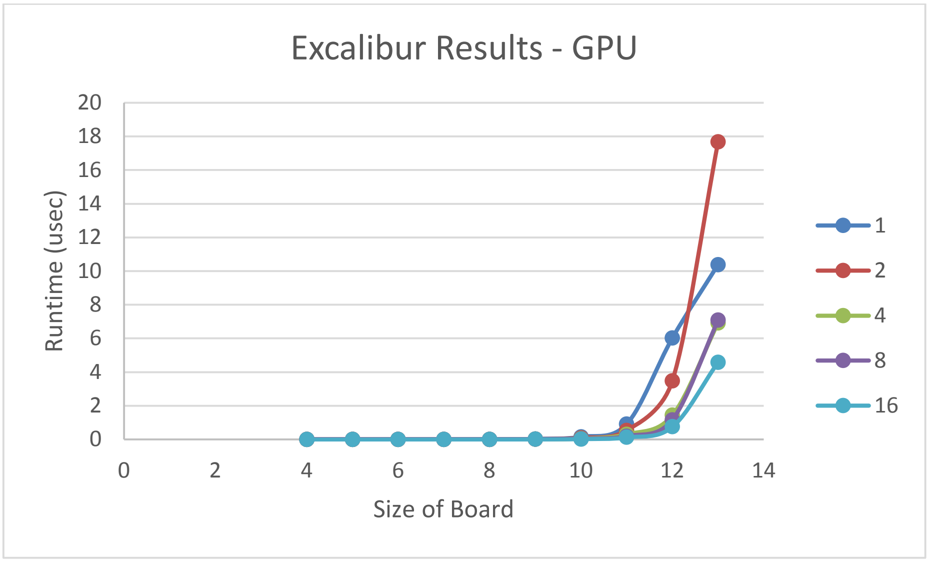


Fig. 5 Nqueens results on Excalibur using the GPU architecture

These results show the importance of autotuning due to the variation of performance one can see through the graphs. In all cases, 16 seems to perform the best, but if that is not feasible for the problem space, one would see a huge variation in results. In addition, this provides a valid way to compare the architectures. We can now compare the best performing kernels at each size with confidence that we are running an optimal kernel for each. This data can now be used to predict performance on each architecture with the highest degree of accuracy. This data also shows how the different numbers of threads impact performance.

8. Conclusions

The work presented here details the steps taken to add a new kernel to the autotuned heterogeneous benchmark suite developed by ARL. The algorithm integrated within this work expands the breadth of areas covered by this suite.

The results show the impact of autotuning and how it can be used to better predict performance of applications on different architectures. The autotuning allows for the most optimal runtimes to be produced, thus enabling consistent comparison among architectures.

9. Future Work

Future work includes integrating the test bench program into the heterogeneous benchmark suite at ARL. This means creating child classes for Nqueens' specific parameters to follow the coding design that has already been established. To provide a better picture of performance on the different architectures more parameters should be added. One of the proposed parameters includes a parameter to force each thread to do more work. Right now, each of the threads updates one position. The parameter would expand this so that threads would update more positions. A maximum number of position updates would have to be determined. This will ensure that the threads have enough work to do, which will help better predict performance.

Other parameters could include changing the workgroup size to vary how much work each work group does and vectorizing the code to use local memory. Each of these will help exercise various elements of the architecture to really help guide programmers as to what is best for each architecture while providing valuable insight into how the architecture works.

Other work outside of this kernel includes adding more parameterized kernels to the benchmark suite. The steps outlined on how the Nqueens algorithm was added will be followed for each new kernel. The more kernels that are added, the more coverage the benchmark suite provides.

10. References

1. Asanovic K, Bodik R, Catanzaro BC, Gebis J, Husbands P, Keutzer K, Patterson D, Plishker W, Shalf J, Williams S. The landscape of parallel computing research: a view from Berkeley. Berkeley (CA): EECS Department, University of California, Berkeley; 2006.

2. Infantolino J, Park S, Shires D. Selecting a benchmark suite to profile high-performance computing (HPC) machines. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Nov. Report No.: ARL-TR-7141.

3. Feng W, Lin H, Scogland T, Zhang J, OpenCL and the 13 dwarfs: a work in progress. Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering; 2012. p. 291–294.

4. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS. The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units; 2010.

5. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization; 2009.

6. Murphy RC, Wheeler KB, Barrett BW, Ang JA. Introducing the graph 500. Cray User Group, Inc.; 2010.

7. Dongarra J, Luszczek P. Linpack benchmark. Encyclopedia of Parallel Computing. Springer; 2011. p. 1033–1036.

8. Dongarra J, Heroux MA. Toward a new metric for ranking high performance computing systems. Sandia National Laboratory (NM);2013. 2013–4744.

9. Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J. From CUDA to OpenCL: towards a performance-portable solution for multi-. Parallel Computing. 2012;38(8):391–407.

10. Fang J, Varbanescu AL, Sips H. A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing; 2011.

11. McIntosh-Smith S, Boulton M, Curran D, Price J. On the performance portability of structured grid codes on many-core computer architectures. The International Conference for High Performance Computing, Networking, Storage, and Analysis; 2014.

12. Abu-Sufah W, Karim AA. Auto-tuning of sparse matrix-vector multiplication on graphics processors. The International Conference for High Performance Computing, Networking, Storage, and Analysis; 2013.

13. Phothilimthana PM, Ansel J, Ragan-Kelley J, Amarasinghe S. Portable performance on heterogeneous architectures. In: ACM SIGPLAN Notices. March 2013;48(4):431–444.

14. Pennycook S, Hammond SD, Wright SA, Herdman J, Miller I, Jarvis SA. An investigation of the performance portability of openCL. Journal of Parallel and Distributed Computing. 2013;73(11):1439–1450.

15. Kaklamanis C, Persiano G. Branch-and-bound and backtrack search on mesh-connected arrays of processors. Mathematical Systems Theory. 1994;27:471–489.

16. Su B-Y, Catanzaro B, Sundaram N. Backtrack and branch-and-bound [accessed 2015 Sep 9]. http://view.eecs.berkeley.edu/wiki /Backtrack_and_Branch-and-Bound.

17. He H, Daume III H, Eisner JM. Learning to search in branch and bound algorithms. In: Advances in Neural Information Processing Systems (NIPS). 2014: 3293–3301.

18. Bader DA, Hart WE, Phillips CA. Parallel algorithm design for branch and bound. In Tutorials on Emerging Methodologies and Applications in Operations Research. Springer New York. 2005:5-1.

19. US Army Research Laboratory (ARL) Dedicated Short-Range Communications (DSRC); 2016. [accessed 2016 Jan 20]. http://www.arl.hpc.mil.

List of Symbols, Abbreviations, and Acronyms

2-D	2-dimensional
API	application program interface
ARL	US Army Research Laboratory
BFS	Breath-First Search
BBB	Backtrack Branch and Bound
CPU	Central Processing Unit
DSRC	Department of Defense Supercomputing Resource Center
FFT	Fast Fourier Transform
GPU	Graphic Processing Units
HPC	High-Performance Computing
HPCG	High-Performance Conjugate Gradient
MIC	Many Integrated Core
MPI	Message Passing Interface
SHOC	Scalable Heterogeneous Computing Benchmark Suite
SPMV	Sparse Matric-Vector Multiplication

1	DEFENSE TECHNICAL
(PDF)	INFORMATION CTR
	DTIC OCA
2	DIRECTOR
(PDF)	US ARMY RESEARCH LAB
	RDRL CIO LL
	IMAL HRA MAIL & RECORDS
	MGMT
1	GOVT PRINTG OFC
(PDF)	A MALHOTRA
1	DIRUSARL
(PDF)	RDRL CIH S
	J INFANTOLINO